Introduction

2017-05-23

Overview

FuzzyR is an extension version of the previous FuzzyToolkitUoN, developed based on the R language called as FuzzyR. The toolkit provides the implementation of design and simulate fuzzy logic systems using Type 1 Fuzzy Logic as previous toolkit includes the new features with the introduction of graphical user interface (GUI) and an adaptive neuro-fuzzy inference system (ANFIS). Produced by the Intelligent Modelling & Analysis Group, University of Nottingham.. For more details on using R Markdown see https://cran.r-project.org/web/packages/FuzzyR/index.html.

R version

Packages need to be installed under this version (>=3.0.0) of R.

Installation

To install the package from CRAN, type:

```
install.packages("FuzzyR")
```

Load the package

Next, load the package:

```
library(FuzzyR)
```

Get started !

Please refer to the tutorials (Laboratory Session) and code samples provided in this package.

Laboratory Session : Part 1

2017-05-23

Overview

In this lab, you will enter sometimes unfamiliar/obscure R syntax into the command window. Many of these commands will be useful to you and so in order to get the best out of the lab sessions, you should try to understand how they work and what they do before moving on.

After this session, you will be able to use R to :

- perform simple matrix calculations
- save data to a file
- load matrices from file
- use the R help function
- draw simple plots using R

Instructions

Double click the R icon to get R up and running. Feel free to peruse the menu-driven help section at your leisure - for now we just want to use the command window and to do some calculations for us.

All the instructions below for some time refer to the command window. The command line starts with > , so type your commands after this symbol.

For the purpose of this worksheet, the commands you should type are in bold italic with the instructions you should follow in regular typeface. Sometimes R will return to you values which are slightly different to those in this worksheet. Do not be concerned about this: it is the nature of using random processes.

1. Variables – declaring and assigning values

In order to enter a variable for use in R, variables can be assigned by using the following syntax. Here *num* is assigned the value of 2.

```
num <- 2
#or
num = 3

# display the value of variable num
num
```

```
## [1] 3
```

2. Variables – *'matrix'*

The R language has its origins in mathematical and statistical computing, and its main type of variable is the 'matrix'. A matrix is a two dimensional list of numbers, similar to a two-dimensional array in other programming

languages, but with properties similar to the mathematical notion of a matrix.

Please type the following matrix input:

```
a <- matrix(c(1,2,3,4), nrow = 2, ncol = 2, byrow=TRUE)
```

where 'matrix' instructs R to accept a matrix, nrow/ncol are the number of rows and columns respectively, and 'byrow' is if the input is by row (true) or by column (false). If you want to find out more about this function, type **help(matrix)**.

Type 'a' and the output should look something like:

```
# display the value of matrix a
a
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

3. R commands use the (<-) and (=) notation

A number of R commands use the <- notation for assignment. However, in nearly all cases, an equals sign (=) will also work. As with many open source packages like this there are often a number of ways to perform the same operation. Have a look at how the following commands can be used to enter a matrix. What are the differences if any?

Please type the following input and observe the output:

```
# create matrix a
a = rbind(c(1,2), c(3,4))
# display the value of matrix a
a
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```
# create matrix a
a = cbind(c(1,2), c(3,4))
# display the value of matrix a
a
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
# create matrix a
a = matrix(c(1,2,3,4), nrow=2)
```

```
# display the value of matrix a
a
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
# create matrix a
a = c(1,2,3,4); dim(a)=c(2,2)
# display the value of matrix a
a
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Now enter another matrix into variable *b*:

```
# create matrix b
b = rbind(c(1,3), c(1,4))
# display the value of matrix b
b
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    1    4
```

4. Multiply and add matrices

We want to check that R knows how to multiply and add matrices correctly. The syntax for adding is simply '+'. The syntax for multiplying whole matrices is %*%.

Please enter the following commands and check the answers:

```
# command 1
a + b
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    3    8
```

```
# command 2
a %*% b
```

```
##      [,1] [,2]
## [1,]    4   15
```

```
## [2,]    6   22
```

Let’s create another new matrix c:

```
# create another matrix
c = rbind(c(1,2,3), c(3,4,5))
# display the value of matrix c
c
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    4    5
```

Recall that we can’t always multiply matrices if they are incompatible in size. Lets try to multiply c*a. Does it return an error?

```
# command 3
c %*% b

# return an error !!!
```

Can we multiply a * c though?

```
# command 4
a %*% c
```

```
##      [,1] [,2] [,3]
## [1,]   10   14   18
## [2,]   14   20   26
```

In order to multiply matrices element-by-element, the notation is different - simply a*a.

Now try the following and make sure you understand the differences between command 5 and 6:

```
# command 5
a * a
```

```
##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

```
# command 6
a %*% a
```

```
##      [,1] [,2]
## [1,]    7   15
## [2,]   10   22
```

Similarly for powers of the numbers in the matrix:

```
# command 7
a^3
```

```
##      [,1] [,2]
## [1,]    1   27
## [2,]    8   64
```

If you want to store the answer, another variable, say y, can be set to be the answer. As R is a programming language, variable assignment is possible in the usual manner.

```
# command 8
y= a%*%c
# display the value of variable y
y
```

```
##      [,1] [,2] [,3]
## [1,]   10   14   18
## [2,]   14   20   26
```

And now you can use y in a calculation:

```
# command 9
y + c(1, 2, 3) + c(1, 2, 5)
```

```
##      [,1] [,2] [,3]
## [1,]   12   22   22
## [2,]   18   22   34
```

R has many built-in capabilities to perform various manipulations on matrices, such as the ability to transpose matrices. This can be performed using the **aperm()** function - this is a useful function with a range of transposition options. However, the quick notation for the plain transpose is :

```
# command 10
t(y)
```

```
##      [,1] [,2]
## [1,]   10   14
## [2,]   14   20
## [3,]   18   26
```

So the transpose flips rows and columns. This is useful in situations where the parameters to a function require a column vector when our data is a row vector.

If we want to add an additional column to an existing matrix we can use the **cbind( )** command. Alternatively we can do this using the matrix editor through the **fix()** command.

```
# command 11
```

```
z= cbind(c, c(1,2))
# display the value of variable z
z
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    1
## [2,]    3    4    5    2
```

5. Get or Set Working Directory

The matrix a is not a big matrix - but it might be and we might want to save it to a file. The first thing to do is to ensure that the current directory is one we can write into and read from. It is best if this is a directory on the **H: drive** (*The type of pathname you use will obviously depend on the operating system you are using*).

We can find which directory is the current working directory with **getwd()** and change it with **setwd()**, or we can use the File menu. Note that the forward slash '/' is used within R as the directory separator, so we might do something like :

```
# set working directory
setwd("H:/Temp")
```

6. Data Output

Now matrix a can be written to a file (*data.txt*). Look at this file to see what you have got.

```
# write the values of variable a into file data.txt.
# data.txt file will be automatically created
write.table(a, file="data.txt")
```

We can load files as well – be careful with the quote marks. They are essential.

```
z= read.table("data.txt")
# display the value of variable z
z
```

```
##   V1 V2
## 1  1  3
## 2  2  4
```

There are also commands which read and write delimited text if you want other than space delimiters. Colon delimited files are used on some unix systems – it is more usual to use commas or tabs on windows systems. Note : In the following syntax, the *a.txt* file is just an example and does not exist in your directory.

```
L= read.table("a.txt", header=FALSE, sep=":")
```

7. R help files – a useful source of information

The R help files are a useful source of information – simply type your keyword into the following syntax:

```
help("plot")
#or
?plot
```

You can also used the keywords:

```
help.search("line")
```

If you want to find out more about the help function then use:

```
help(help)
```

If you want to keep track of what you are doing try the history command (use the help to find out about it).

8. Save and Exit R program

If you want to keep all your working you can save the workspace either through the application (under 'workspace') or through the command prompt – again see help on save for the details.

To exit the program using the command line use:

```
quit()
```

9. Entering data

Entering data by hand can be tedious so R allows creating uniformly spaced data points :

```
a= 0:100
# display the value of variable a
a
```

```
##   [1]   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
##  [18]  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33
##  [35]  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
##  [52]  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67
##  [69]  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
##  [86]  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Sequences of non-integer numbers can be done using this:

```
b= seq(0,1,.01)
# display the value of variable b
b
```

```
##   [1] 0.00 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13
```

```
##  [15] 0.14 0.15 0.16 0.17 0.18 0.19 0.20 0.21 0.22 0.23 0.24 0.25 0.26 0.27
##  [29] 0.28 0.29 0.30 0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.40 0.41
##  [43] 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49 0.50 0.51 0.52 0.53 0.54 0.55
##  [57] 0.56 0.57 0.58 0.59 0.60 0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69
##  [71] 0.70 0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.80 0.81 0.82 0.83
##  [85] 0.84 0.85 0.86 0.87 0.88 0.89 0.90 0.91 0.92 0.93 0.94 0.95 0.96 0.97
##  [99] 0.98 0.99 1.00
```

10. Creating a plot

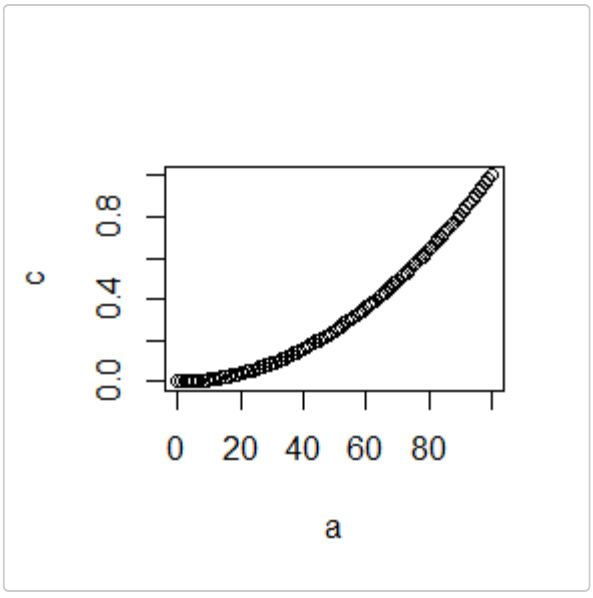
You have probably realized by now that the output of assignments is automatically suppressed. However, just type ‘b’ now to show you the variable. Now, just make something to plot, e.g.:

```
c= b ^ 2
# display the value of variable c
c
```

```
##   [1] 0.0000 0.0001 0.0004 0.0009 0.0016 0.0025 0.0036 0.0049 0.0064 0.0081
##  [11] 0.0100 0.0121 0.0144 0.0169 0.0196 0.0225 0.0256 0.0289 0.0324 0.0361
##  [21] 0.0400 0.0441 0.0484 0.0529 0.0576 0.0625 0.0676 0.0729 0.0784 0.0841
##  [31] 0.0900 0.0961 0.1024 0.1089 0.1156 0.1225 0.1296 0.1369 0.1444 0.1521
##  [41] 0.1600 0.1681 0.1764 0.1849 0.1936 0.2025 0.2116 0.2209 0.2304 0.2401
##  [51] 0.2500 0.2601 0.2704 0.2809 0.2916 0.3025 0.3136 0.3249 0.3364 0.3481
##  [61] 0.3600 0.3721 0.3844 0.3969 0.4096 0.4225 0.4356 0.4489 0.4624 0.4761
##  [71] 0.4900 0.5041 0.5184 0.5329 0.5476 0.5625 0.5776 0.5929 0.6084 0.6241
##  [81] 0.6400 0.6561 0.6724 0.6889 0.7056 0.7225 0.7396 0.7569 0.7744 0.7921
##  [91] 0.8100 0.8281 0.8464 0.8649 0.8836 0.9025 0.9216 0.9409 0.9604 0.9801
## [101] 1.0000
```

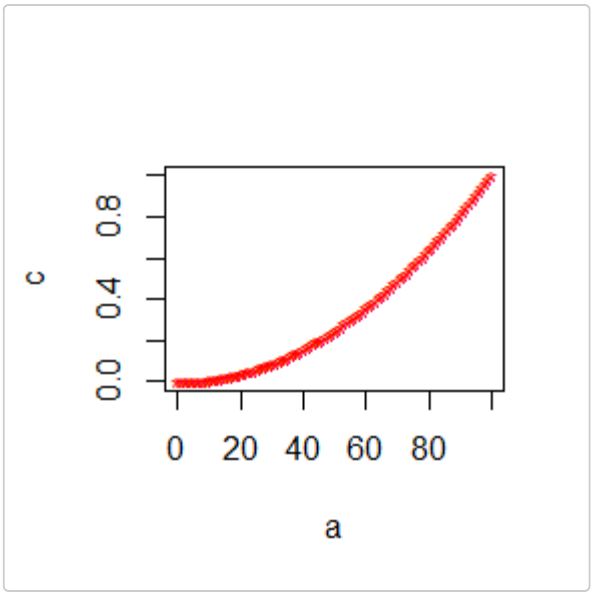
R comes with built in graphics capability and is a useful tool for investigating data and for plotting advanced graphs and charts, including a wide variety of 3D plots.

```
plot(a, c)
```



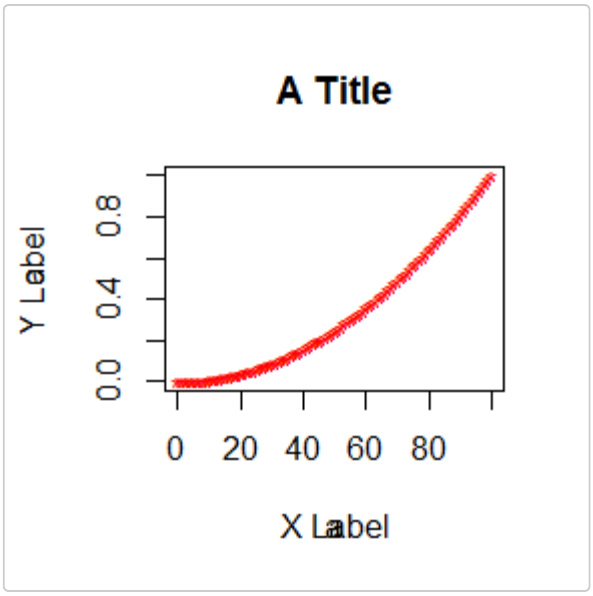
See the help on plot to see the various variants of plotting. For example, see the effect of the following: plots.

```
plot(a, c, col="red", pch="*")
```



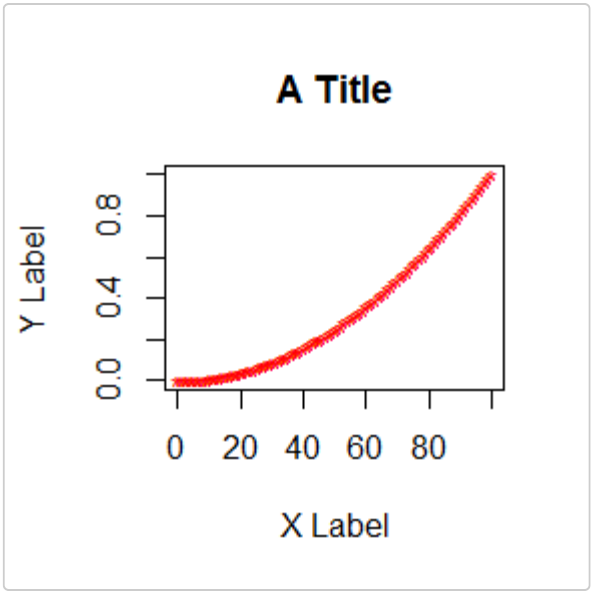
While the graphical window is still open, then add a title and axes labels using the following command.

```
plot(a, c, col="red", pch="*")
title(main = "A Title", xlab = "X Label", ylab = "Y Label")
```



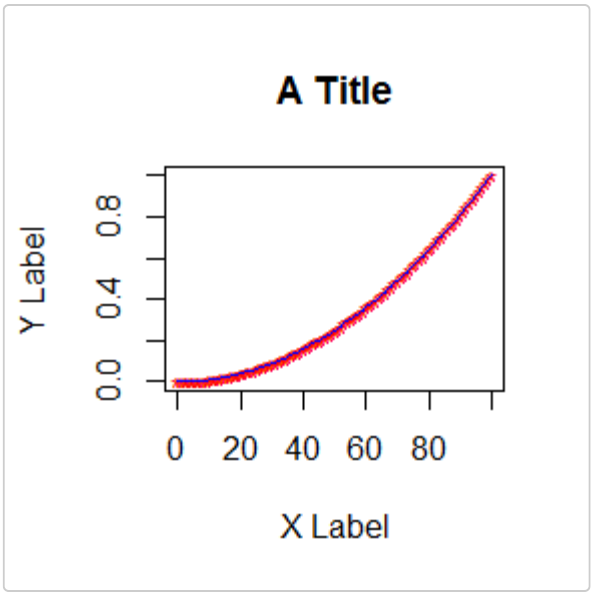
You can actually do this all within the plot command.

```
plot(a, c, col="red", pch="*", main = "A Title", xlab = "X Label", ylab = "Y Label")
```



We can use the lines command to plot more than one data series on one figure (or, as in this case, add a second set of symbols for the same data series):

```
plot(a, c, col="red", pch="*", main = "A Title", xlab = "X Label", ylab = "Y Label")
lines(a, c, col="blue")
```



Laboratory Session : Part 2

2017-05-23

Overview

In this session you will :

- learn how to access columns and rows of data
- how to do basic 2D plots
- how to use control structures
- meet the idea of an r-file.

Instructions

Ensure that you have a datafile, **“a.txt”** containing the following matrix:

A	B
2	4
3	5
4	6

1. Read data – read.table

Use the read.table command to get the data from the file data.txt that you created last time into a matrix called ab. Check that the data has been read correctly.

```
ab = read.table("a.txt", header = T)
# display the value of variable ab
ab
```

```
##   A B
## 1 2 4
## 2 3 5
## 3 4 6
```

To get the second column of ab try:

```
ab$B
```

```
## [1] 4 5 6
```

```
ab[2]
```

```
##   B
## 1 4
## 2 5
## 3 6
```

```
ab[,2]
```

```
## [1] 4 5 6
```

To get the second row of ab try:

```
ab[2,]
```

```
##   A B
## 2 3 5
```

Set the vector a to be the first column of ab.

```
a= ab$A
# display the value of variable a
a
```

```
## [1] 2 3 4
```

If you just want to focus on this matrix and avoid the **$** notation, then you can attach the data, so the variables can be accessed directly

```
attach(ab)
# display the value of column in ab without use $, just enter the column's name.
A
```

```
## [1] 2 3 4
```

When you're done with that matrix/dataset, you can try this:

```
detach(ab)
# display the value of column in ab, you need to use $ notation.
ab$A
```

```
## [1] 2 3 4
```

2. Add column

Add a column to ab – described earlier (hint – you need to cbind or fix the matrix)

```
# Example of using -- cbind
# Add a new column C with values; 5,7,9.
ab = cbind(ab, C=c(5,7,9))
# display the value of variable ab
ab
```

```
##   A B C
## 1 2 4 5
## 2 3 5 7
## 3 4 6 9
```

3. For-loops and If..else

R has standard programming features like for-loops which can be used in a variety of ways. Here we also introduce **rnorm()** which is a way of generating normally distributed sequences of numbers in a random fashion. Also, note the notation for accessing separate elements of the matrix, using the square braces:

```
for(i in 1:nrow(ab))
{
f = 0.5 + rnorm(ncol(ab));
ab[i,] = ab[i,]+f
}
```

In order to get this to work, press return at the end of each line (which should return a '+' symbol). Don't forget a semicolon between each line, and remember to add curly braces. This piece of code has altered the rows of ab by adding random elements to them. R also has if statements:

```
for(i in 1:nrow(ab))
{
f = 0.5 + rnorm(ncol(ab));
ab[i,] = ab[i,]+f

    if ( i == 1 ) {
        print('five!')
    } else {
        print('not five!')
    }

}
```

```
## [1] "five!"
## [1] "not five!"
## [1] "not five!"
```

```
for(i in 1:nrow(ab))
{
f = 0.5 + rnorm(ncol(ab));
ab[i,] = ab[i,]+f
```

```
    if ( i == 2 ) {
        print('five!')
    } else {
        print('not five!')
    }

}
```

```
## [1] "not five!"
## [1] "five!"
## [1] "not five!"
```

```
for(i in 1:nrow(ab))
{
f = 0.5 + rnorm(ncol(ab));
ab[i,] = ab[i,]+f

    if ( i == 3 ) {
        print('five!')
    } else {
        print('not five!')
    }

}
```

```
## [1] "not five!"
## [1] "not five!"
## [1] "five!"
```

4. Saving your data

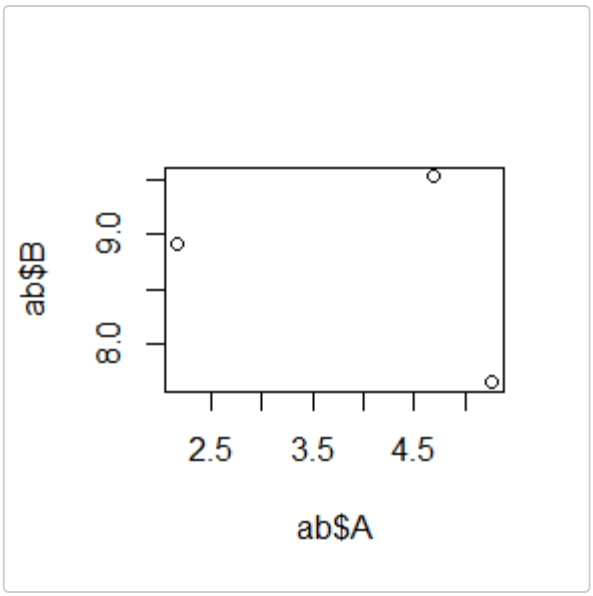
There are many other language constructs which you can use for programming – see the help for syntax. Now save your new version of ab in **data.txt** – you will be using this data later on.

```
write.table(ab, file="data2.txt")
```

5. Creating a plot

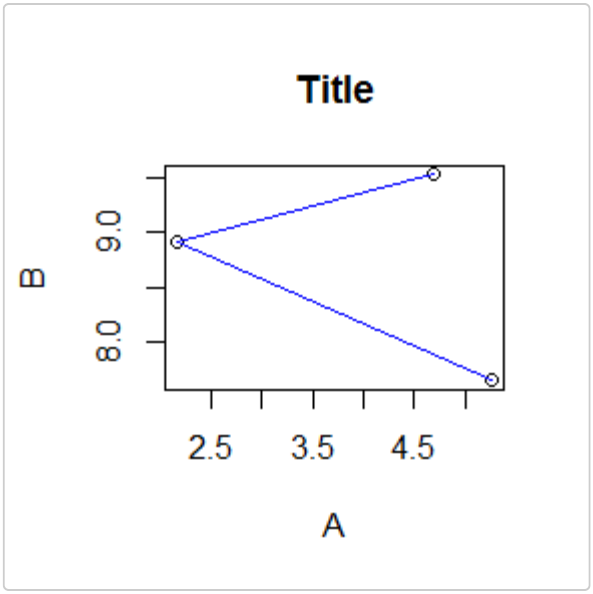
We can plot the y-values against the x-values on one graph like this :

```
plot(ab$A, ab$B)
```



We can use the lines command to plot more than one data series on one figure:

```
plot(ab$A, ab$B, main = "Title", xlab = "A", ylab="B")
lines(ab$A, ab$B, col="blue")
```



6. Function files

A function file allows you to repeat a series of commands that you have created without having to type them all in again. This is particularly useful for conducting experiments where you want to repeat tests with only minor changes. Almost all R commands are actually function files. Create a new source file by choosing File|New from the menu enter the following:

```
myplot <-function(){
    pdf("plot1.pdf")
    a = seq(0,1,0.1)
    c = a * (a^2)
    plot(a, c, col="blue", pch="*")
```

```
    dev.off()
}
```

Save this as **myplot.r**. You can add this file to the project through the interface or by using the **source()** command:

```
source("myplot.r")
```

You will now be able to enter **myplot()** at the command line and see the effect (note that this file must be in the correct directory).

Now edit the file, so that it contains:

```
myplot <-function(name){
    pdf(paste0(name, ".pdf"))
    a = seq(0,1,0.1)
    c = a * (a^2)
    plot(a, c, col="blue", pch="*")
  dev.off()
}
```

Every time you change the file, you should save it, and then you must re-source it, so that the new contents are loaded into R. So, again enter:

```
source("myplot.r")
```

And then enter:

```
myplot("test")
```

The **'name'** variable is now an argument to the function **'myplot'**. R is a weakly typed language. This means that you do not have to specify what type (integer, string, etc.) of variable that **'name'** is; it will simply contain whatever is passed to it on the command line. It is up to you to make sure that you use a variable inside a function in an appropriate manner. In our example, 'name' is taken to be a string (character) variable that will contain the filename. We add the ".pdf" part to the filename, so that the user does not have to type this each time – see **help(paste)** for information on **'paste'** and **'paste0'**.

Note that, as in most programming languages, you can pass multiple arguments to a function. Change the file so that it contains:

```
myplot <-function(name, colour){
    pdf(paste0(name, ".pdf"))
    a = seq(0,1,0.1)
    c = a * (a^2)
    plot(a, c, col=colour, pch="*")
    dev.off()
}
```

Again, save and re-source the changed file, and then try:

```
source("myplot.r")
```

```
myplot("test-blue", "blue")
myplot("test-red", "red")
```

Laboratory Session : Part 3

2017-05-23

Overview

In this session you will do more with data analysis functions, learning:

- how to load and manipulate data
- how to describe data, including statistical summaries and cross-tabulations
- how to do more advanced types of plotting, including multiple plots
- how to do basic 3D plotting

Instructions

You can download data files to experiment with from the UCI Machine Learning Repository: http://archive.ics.uci.edu/ml

R also has several built-in data sets available, including the famous ***'iris'** data. In order to see a list of in-built data, please type:

```
    data()
```

1. Load data – Iris

In this section, we will use the iris data, but will copy it into our own variable, **ir**, so that we can change names, etc.:

```
# load data iris into variable ir
ir= iris
```

Technically, the type of the variable, or 'class' as it is known in R terminology, is not a matrix, but a data.frame:

```
class(ir)
```

```
## [1] "data.frame"
```

However, it broadly behaves in a similar way to a 2D matrix. For detailed differences, see R help and online manuals.

2. Data manipulation – Iris

Once the data is in such a variable, we can set, view and use the names of each of the columns, rather than using the numeric column indices:

```
names(ir) = c("sepal.length", "sepal.width",    "petal.length", "petal.width", "class")
names(ir)
```

```
## [1] "sepal.length" "sepal.width"  "petal.length" "petal.width"
## [5] "class"
```

Viewing data at column; sepal.length :

```
ir$sepal.length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Of course, the numeric indices can still be used:

```
# view ir data : row 1
ir[1,]
```

```
##   sepal.length sepal.width petal.length petal.width  class
## 1          5.1         3.5          1.4         0.2 setosa
```

```
# view ir data : row 1 ~ 3
ir[1:3,]
```

```
##   sepal.length sepal.width petal.length petal.width  class
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3.0          1.4         0.2 setosa
## 3          4.7         3.2          1.3         0.2 setosa
```

```
# view ir data : row 1, 51, 101
ir[c(1,51,101),]
```

```
##     sepal.length sepal.width petal.length petal.width      class
## 1            5.1         3.5          1.4         0.2     setosa
## 51           7.0         3.2          4.7         1.4 versicolor
## 101          6.3         3.3          6.0         2.5  virginica
```

Have a play with other indexing to understand the capabilities of R.

3. Describe data : statistical summaries and cross-tabulations

There are various functions to calculate statistical properties of data, such as:

```
mean(ir$sepal.length)
```

```
## [1] 5.843333
```

```
median(ir$petal.width)
```

```
## [1] 1.3
```

```
summary(ir)
```

```
##   sepal.length    sepal.width     petal.length    petal.width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        class
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

And functions to perform tabulations and cross tabulations:

```
table(ir$class)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

```
ir$sepal.length.class = ifelse(ir$sepal.length < 6,     "setosa", "other")
table(ir$sepal.length.class)
```

```
##
##  other setosa
##     67     83
```

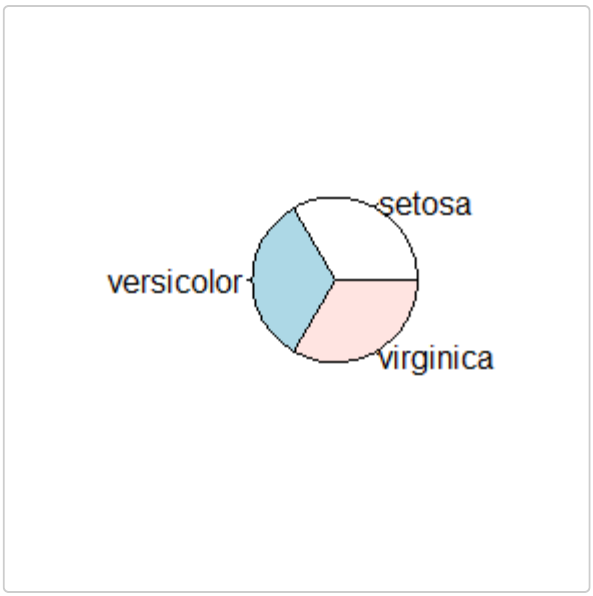
```
ir$sepal.length.class = ifelse(ir$sepal.length < 6,     "setosa", "other")
table(ir$sepal.length.class, ir$class)
```

```
##
##          setosa versicolor virginica
##   other       0         24        43
##   setosa     50         26         7
```

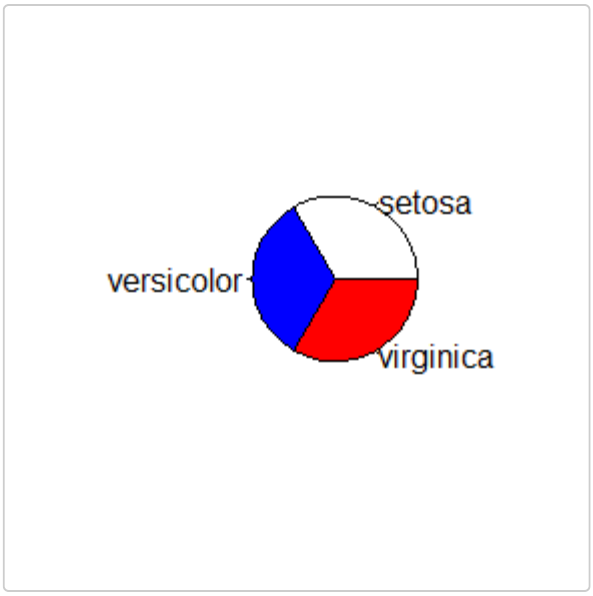
4. Advanced types of plotting

R has many functions for more advanced graphs and plots, for example:

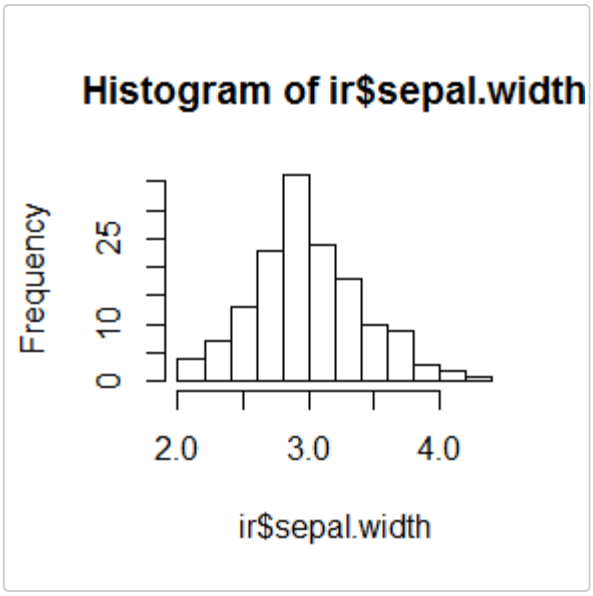
```
pie(table(ir$class))
```



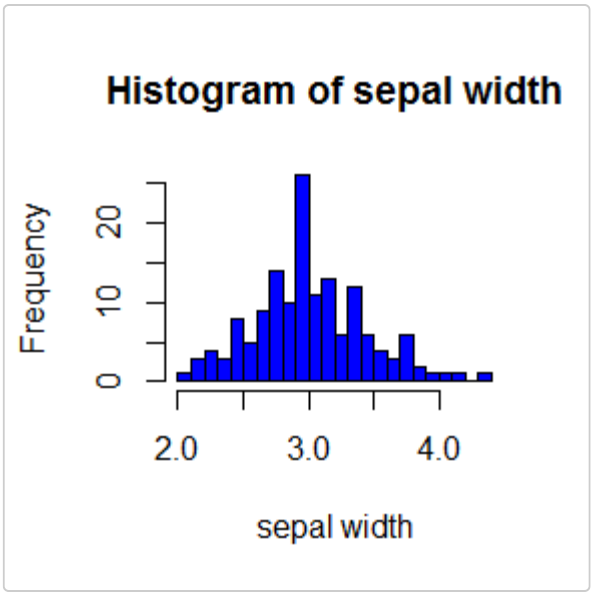

```
pie(table(ir$class), col=c("white","blue","red"))
```



```
hist(ir$sepal.width)
```

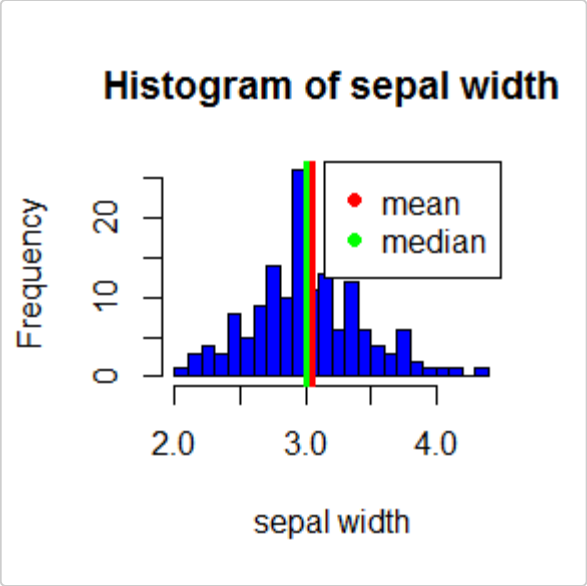



```
hist(ir$sepal.width, breaks=30, col="blue",
    main="Histogram of sepal width", xlab="sepal width")
```



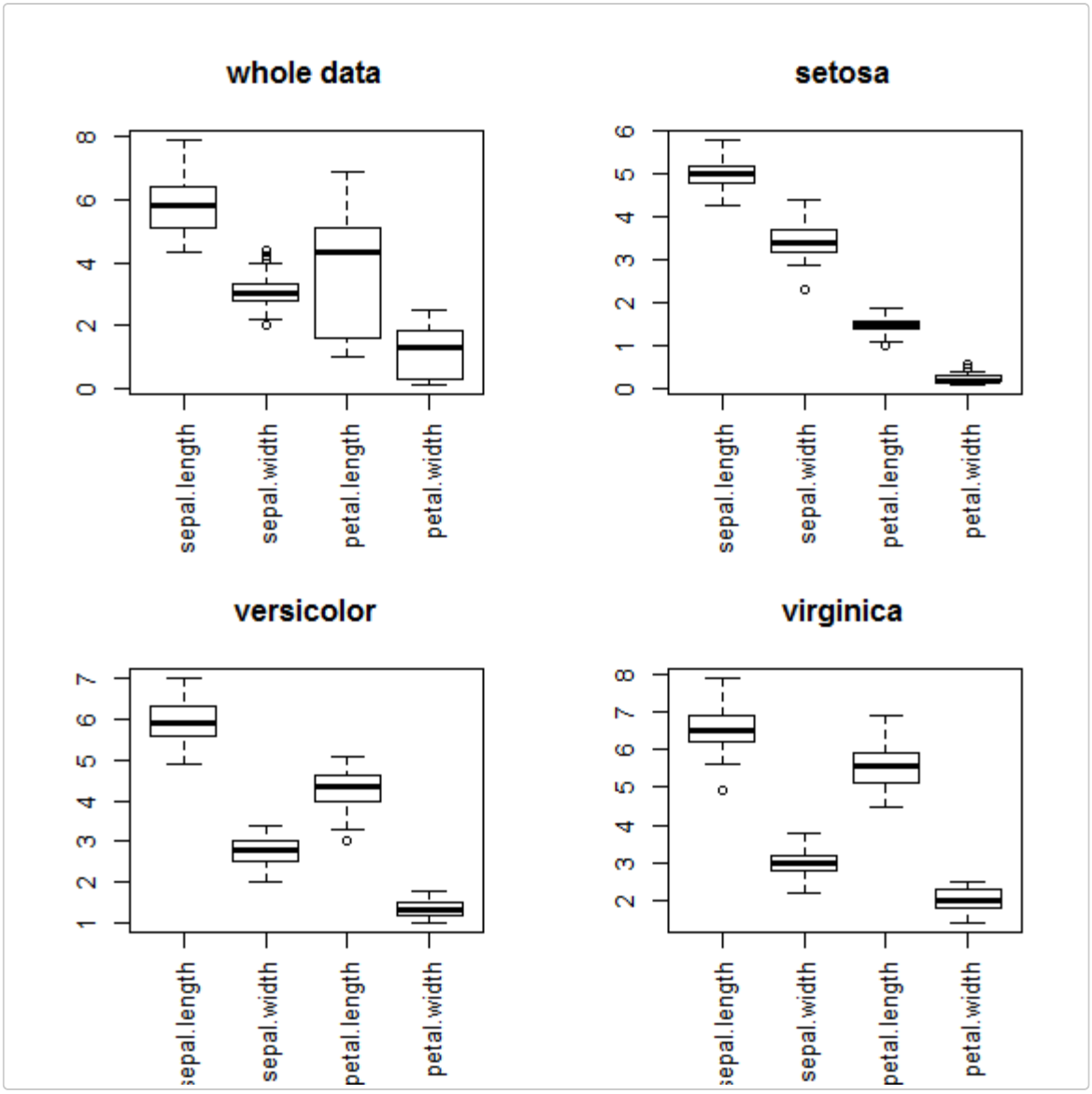
Also, the capability to add extra lines, legends, text, etc. to plots.

```
hist(ir$sepal.width, breaks=30, col="blue", main="Histogram of sepal width", xlab="sepal width")
abline(v=mean(ir$sepal.width),col="red", lwd=3)
abline(v=median(ir$sepal.width),col="green", lwd=3)
legend("topright",c("mean","median"),col=c("red","green"),
    pch=16)
```



R can easily plot multiple plots on one **'window'**:

```
par(mfrow=c(2,2))
boxplot(ir[,1:4], las=3, main="whole data")
boxplot(ir[ir$class=="setosa",1:4], las=3,
    main="setosa")
boxplot(ir[ir$class=="versicolor",1:4], las=3,
    main="versicolor")
boxplot(ir[ir$class=="virginica",1:4], las=3,
    main="virginica")
```



5. An interactive 3-dimensional plotting

Finally, the latest versions of R have introduced interactive 3-dimensional plotting capabilities.

To install the package from CRAN, type:

```
install.packages("rgl")
```

Then, try this :

```
library(rgl)
plot3d(ir$sepal.length,ir$sepal.width,ir$petal.length, col=rainbow(1000),
     xlab="sl",ylab="sw",zlab="pl", size=8)
```

Laboratory Session : Part 4

2017-05-23

Overview

In this session you will practice the use of R by creating a tipper system that does not rely on fuzzy logic:

- How to create a complex function for the tipper problem in R
- Illustrate the effect of your tipper function by plotting a 3-d surface

Instructions

We are going to work through the *'Fuzzy vs. Non-Fuzzy'* tutorial that you met before, but this time actually implementing the non-fuzzy tipper in R. The description is in the *'fuzzy-approach'* worksheet.

1. Create the service and food variables

First of all, let's create the **service** and **food** variables, which each range from 0 to 10 in steps of 0.1:

```
service = seq(0, 10, 0.5)
food = seq(0, 10, 0.5)
# display the value of variable service
service
```

```
##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5
## [15]  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

```
# display the value of variable food
food
```

```
##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5
## [15]  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

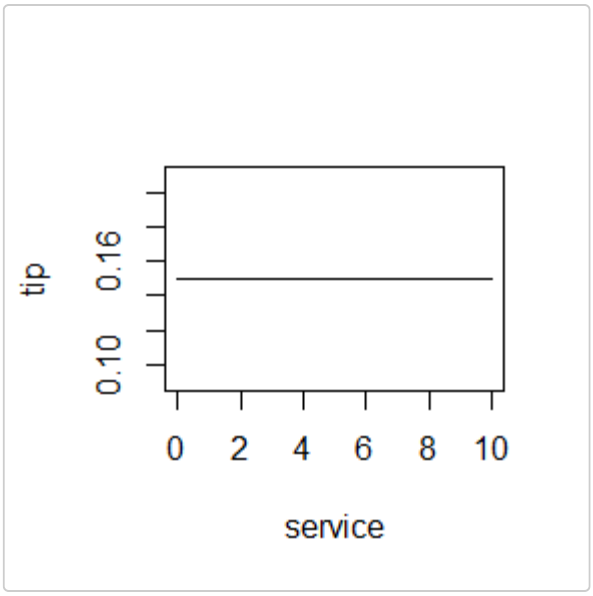
2. Create a plot

Now, working from the worksheet, let's create a fixed **tip** of 15% (0.15) and try to create a plot of the **tip** (on the y-axis) against the **service** (on the x-axis). Naively, we might think this will work:

```
tip = 0.15
plot(service, tip, type= 'l')

# return an error !!!
```

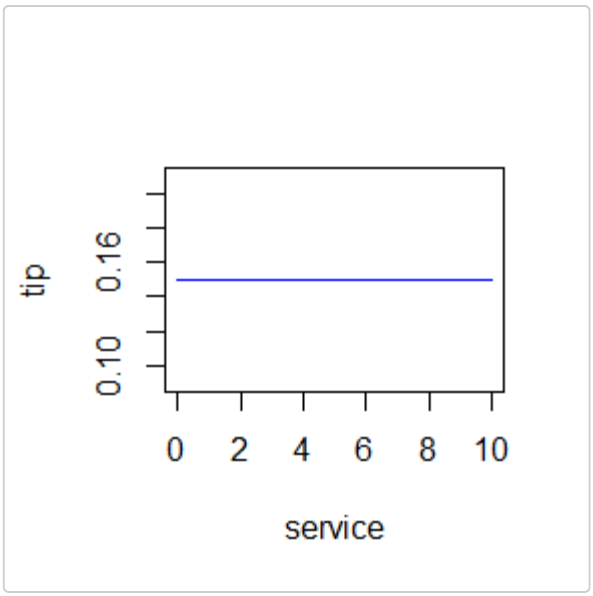
Actually, the plot command is not quite correct, because we are trying to plot a single fixed number (the **tip**) against a range (**service** from 0 to 10). The simplest was of getting around this for now is to use a trick in R:

```
tip= 0.15
plot(cbind(service, tip), type= 'l')
```



We can make it look even more like the MATLAB equivalent by colouring the line blue:

```
plot(cbind(service, tip), type= 'l', col='blue')
```



3. Update variable tip – use an equation

This relationship does not take into account the quality of the **service**, so you need to add a new term to the equation. Because service is rated on a scale of 0 to 10, you might have the **tip** go linearly from 5% if the **service** is bad to 25% if the **service** is excellent. You can achieve this will the following formula:

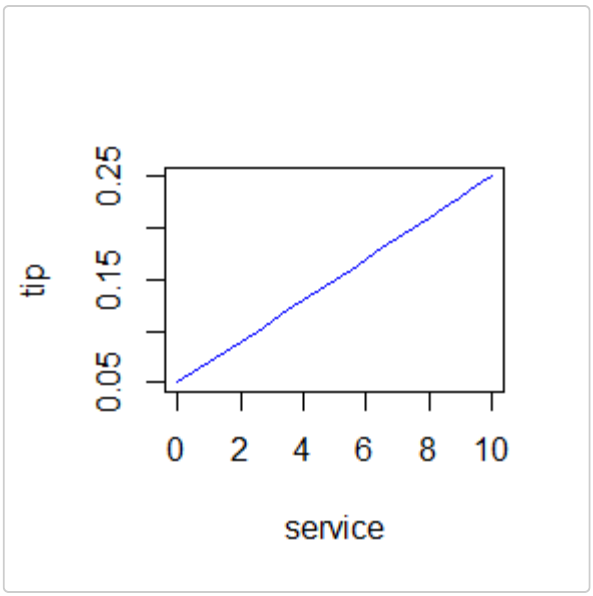
```
tip = 0.20/10 * service + 0.05
```

```
# display the value of variable tip
tip
```

```
##  [1] 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18
## [15] 0.19 0.20 0.21 0.22 0.23 0.24 0.25
```

And (now the **tip** variable is the same length as the **service** variable) the relationship can be seen in the following plot:

```
plot(service, tip, type= 'l', col='blue')
```



4. The Extended Tipping Problem.

The formula does what you want it to do, and is straightforward. However, you may want the **tip** to reflect the quality of the **food** as well. This extension of the problem is defined as follows.

The Extended Tipping Problem. Given two sets of numbers between 0 and 10 (where 10 is excellent) that respectively represent the quality of the **service** and the quality of the **food** at a restaurant, what should the **tip** be?

First, we need to create a function to map **service** and **food** to **tip**, We do this by:

```
sf= function(s,f) 0.20/20 * (s + f) + 0.05
```

And then, we calculate this function over a matrix of **service** and **food** values by:

```
tip = outer(service, food, sf)
```

To see this relationship, we need to plot a 3D graph of tip against both service and food. This can be done in R as follows: Then, try this :

```
library(rgl)
persp3d(service, food, tip, col=tip*40)
```

You must enable Javascript to view this page properly.

You can now extend this function along the same lines as described in the MATLAB tutorial, simply by extending the definition of **'sf'**. I recommend doing this by defining a proper multi-line function in a text editor, cuting and pasting into R as you go. So, for example:

```
library(rgl)

sf= function(s,f) {
    servRatio= 0.8;
    t= servRatio * (0.2/10*s + 0.05) +
        (1 - servRatio) * (0.2/10*f + 0.05)
}
tip = outer(service, food, sf)
persp3d(service, food, tip, col=tip*40)
```

You must enable Javascript to view this page properly.

See if you can extend it to the full complex function at the end of the tutorial.

5. Note – older versions of R

Note that in older versions of R, rather than:

```
tip = outer(service, food, sf)
```

It was necessary to use another R trick, the Vectorize function, as in:

```
tip = outer(service, food, Vectorize(sf))
```

Laboratory Session : Part 5

2017-05-23

Overview

In this session you will learn how to use the **FuzzyR** Toolbox

Instructions

To get access to the Toolbox, simply launch the R interpreter (a shortcut to which should be located on your desktop) and type in the following commands (you will be asked to select a mirror for download after the first command, any will do).

1. Installation

To install the package from CRAN, type:

```
install.packages("FuzzyR")
```

2. Load the package

```
library(FuzzyR)
```

3. Source files

You will need to put two source files in your working directory (FuzzyMF.R and FuzzyR.patch.R) then issue the following commands.

```
source("FuzzyMF.R")
source("FuzzyR.patch.R")
```

4. All the commands

To see all the commands in the toolbox, simply type the following command:

```
ls("package:FuzzyR")
```

```
##  [1] "addmf"                  "addrule"
##  [3] "addvar"                 "anfis.L1.eval"
```

```
##  [5] "anfis.L2.eval"           "anfis.L2.which"
##  [7] "anfis.L3.eval"           "anfis.L4.eval"
##  [9] "anfis.L4.mf.eval"        "anfis.L5.eval"
## [11] "anfis.LI.eval"           "anfis.builder"
## [13] "anfis.dE.dO1"            "anfis.dE.dO2"
## [15] "anfis.dE.dO3"            "anfis.dE.dO4"
## [17] "anfis.dE.dO5"            "anfis.dE.dP1"
## [19] "anfis.dE.dP1.gbellmf"    "anfis.dE.dP1.it2gbellmf"
## [21] "anfis.dE.dP4"            "anfis.dMF.dP.gbellmf"
## [23] "anfis.dO2.dO1"           "anfis.dO3.dO2"
## [25] "anfis.dO4.dO3"           "anfis.dO5.dO4"
## [27] "anfis.eval"              "anfis.optimise"
## [29] "anfis.plotmf"            "anfis.tipper"
## [31] "defuzz"                  "evalfis"
## [33] "evalmf"                  "evalmftype"
## [35] "fis.builder"             "fuzzy.firing"
## [37] "fuzzy.optimise"          "fuzzy.t"
## [39] "fuzzy.tconorm"           "fuzzy.tnorm"
## [41] "fuzzyr.accuracy"         "fuzzyr.match.fun"
## [43] "gbell.fuzzification"     "gbellmf"
## [45] "genmf"                   "gensurf"
## [47] "km.da"                   "linearmf"
## [49] "newfis"                  "plotmf"
## [51] "readfis"                 "showGUI"
## [53] "showfis"                 "showrule"
## [55] "singleton.fuzzification" "singletonmf"
## [57] "tipper"                  "tipperGUI"
## [59] "tipperGUI2"              "x.fuzzification"
```

5. Access documentation – any of these functions

To then access documentation on any of these functions, you need only put a question mark (?) before the function name, for example:

```
?addmf
#or
help(addmf)
```

Which will launch the online documentation page for that specific function. To see all of the documentation in one place in PDF format, you can visit this link:

https://cran.r-project.org/web/packages/FuzzyR/FuzzyR.pdf

6. Draw some membership functions

Now let's draw some membership functions. The **FuzzyR** toolbox has a number of functions available (soon to be expanded upon!) :

- **gbellmf** – generalised bell-shaped membership function
- **gaussmf** – gaussian curve membership function (this is a popular membership function)
- **trapmf** – trapezoidal shaped membership function (this is a popular membership function)
- **trimf** – triangular shaped membership function (popular)

All of these membership functions take various parameters. Try the following code:

```
mytrimf <- genmf('trimf', c(1,2,3))
```

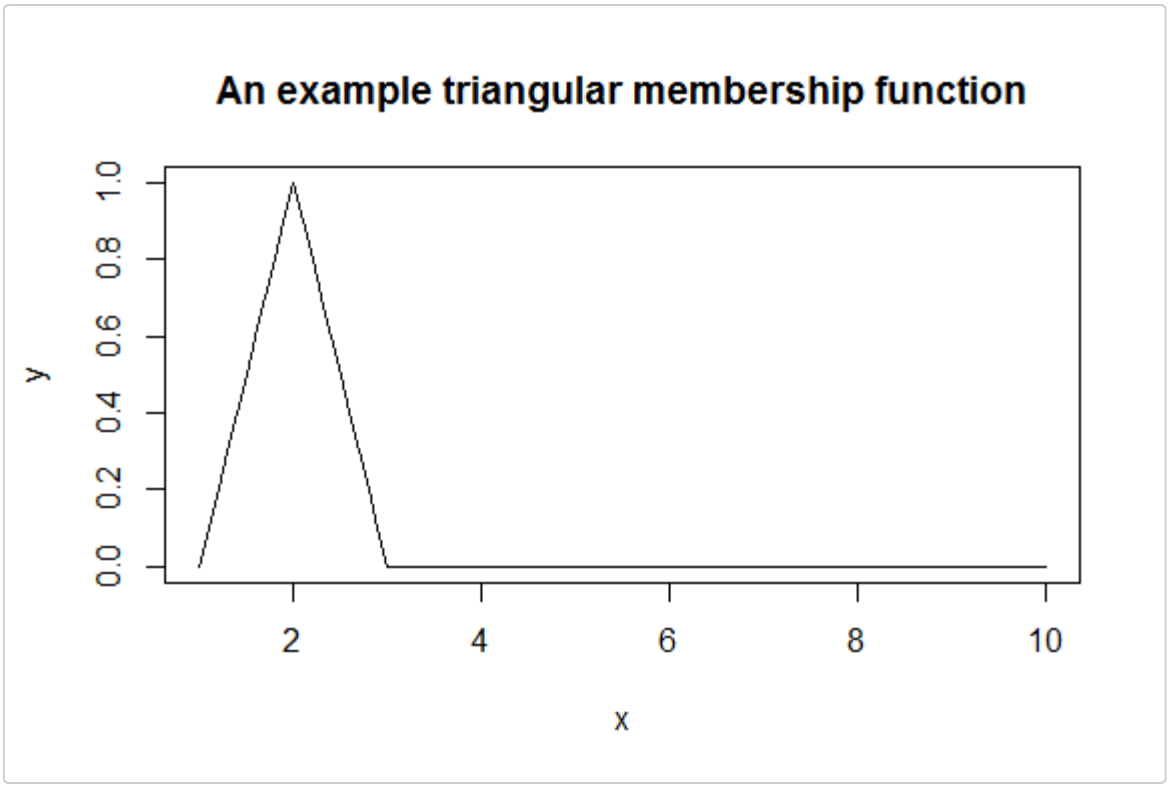
The three parameters (1,2,3) determine the points at which the triangle joins the x axis and the peak. **genmf** is a function to create a membership function

Experiment with some others – especially **gaussmf** and **trapmf**, e.g.:

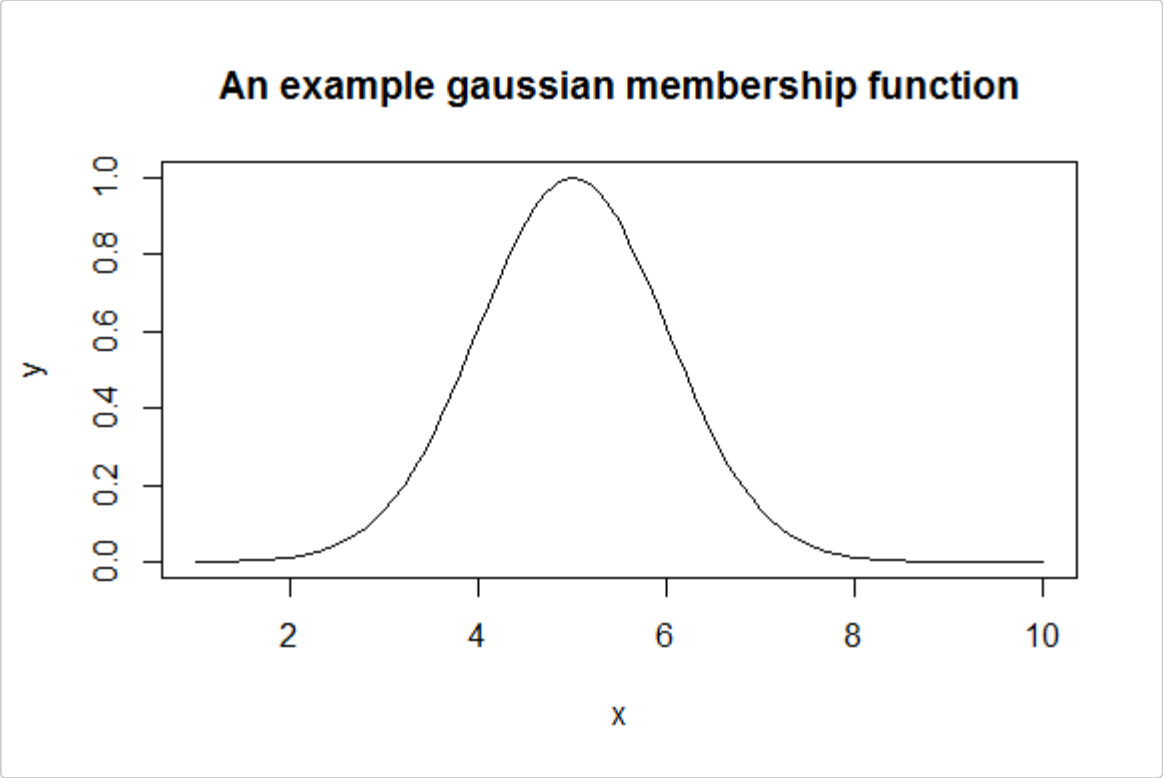
```
mygaussmf <- genmf('gaussmf', c(1,5))
```

where, 1 represents the standard deviation and 5 the centre.

```
x <- seq(1,10,by=0.1)
y <- evalmf(x, mytrimf)
plot(x,y, type='l')
title("An example triangular membership function")
```




```
x <- seq(1,10,by=0.1)
y <- evalmf(x, mygaussmf )
plot(x,y, type='l')
title("An example gaussian membership function")
```



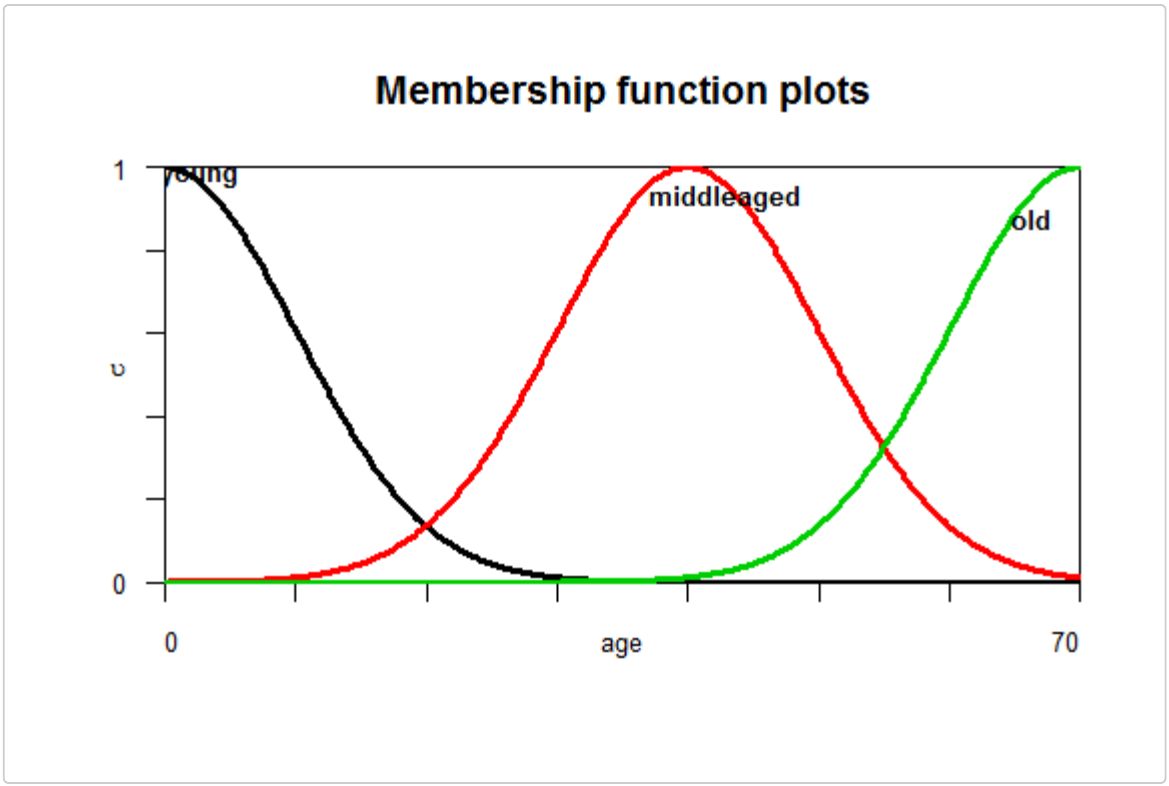
Try different membership functions, with different parameters, to understand their effects.

7. Developing a fuzzy inference system (FIS)

When you build a fuzzy system in R consisting of rules, membership functions and so on you develop a fuzzy inference system – known as an fis in R. To be able to consider membership functions properly, we place everything inside an fis. The following code creates a linguistic variable representing ‘age’ with 3 associated linguistic terms or labels. We see in this code that to do this we create a fis first. Note the use of # to allow for comments. Enter this code into a function file and see the results. Where necessary, refer to the documentation to fully understand what is going on.

```
# This is a simple function to create a linguistic variable
# for age with 3 linguistic labels - young, middleaged and old

# the first statement creates a new fis with string name test
# and R variable a. The fis takes certain defaults which you can
# ignore for the moment.
fis <- newfis('Test');


# This adds a variable called 'linguistic age' to the fis which is
# of type input and lies between 0 and 70
fis <- addvar(fis,'input', 'age',c(0, 70));


# We now add three membership functions - read the help to understand
# the parameters
fis <- addmf(fis, 'input', 1, 'young', 'gaussmf', c(10, 0));
fis <- addmf(fis, 'input', 1, 'middleaged', 'gaussmf', c(10, 40));
fis <- addmf(fis, 'input', 1, 'old', 'gaussmf', c(10, 70));
```

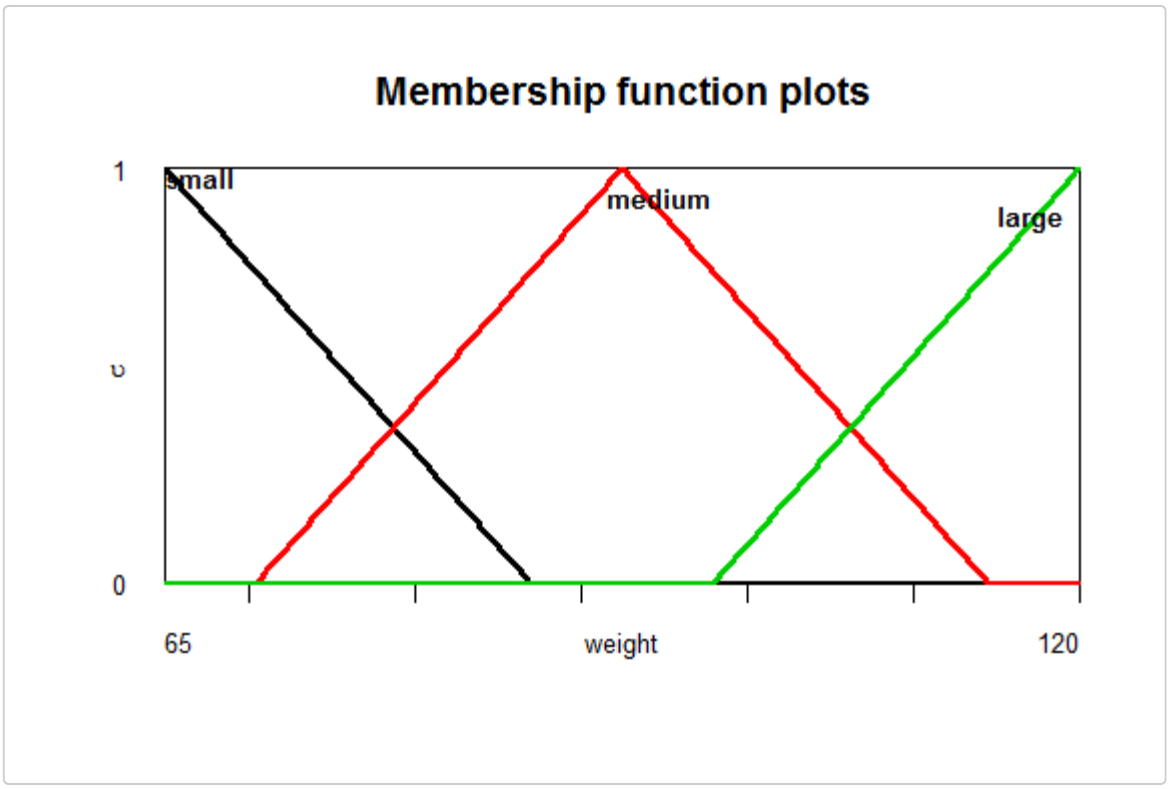
```
# This plots the membership functions
plotmf(fis,'input',1, main = 'Membership function plots')
```



Now add a new variable into your r-file called ‘male weight’ which describes someone as small, medium or large over a domain ranging from 65Kg to 120Kg. Use triangular membership functions. Plot the result.

```
# This is a simple function to create a linguistic variable
# for age with 3 linguistic labels - young, middleaged and old

# the first statement creates a new fis with string name test
# and R variable a. The fis takes certain defaults which you can
# ignore for the moment.
fis <- newfis('Test');


# This adds a variable called 'linguistic age' to the fis which is
# of type input and lies between 0 and 70
fis <- addvar(fis,'input', 'age',c(0, 70));
# A new variable called ‘male weight’ which  ranging from 65Kg to 120Kg.
fis <- addvar(fis,'input', 'weight',c(65, 120));


# We now add three membership functions - read the help to understand
# the parameters
fis <- addmf(fis, 'input', 1, 'young', 'gaussmf', c(10, 0));
fis <- addmf(fis, 'input', 1, 'middleaged', 'gaussmf', c(10, 40));
fis <- addmf(fis, 'input', 1, 'old', 'gaussmf', c(10, 70));

# We now add three membership functions - small, medium or large
fis <- addmf(fis, 'input', 2, 'small', 'trimf', c(43, 65, 87));
```

```
fis <- addmf(fis, 'input', 2, 'medium', 'trimf', c(70.5, 92.5, 114.5));
fis <- addmf(fis, 'input', 2, 'large', 'trimf', c(98, 120, 142));

# This plots the membership functions
plotmf(fis,'input',2, main = 'Membership function plots')
```



Finally add a third variable to the same r-file ‘male feet size’ choosing five sensible labels and domain. Use membership functions that are not Gaussian or triangular. **Please refer to the above examples**.

Pictures are very important when developing fuzzy systems. There is a command to show all three variables in together on one single plot window. First set the window to accept 3-subplots in a column:

```
par(mfrow=c(3,1))
```

Then issue the **plotmf** commands for each of the three variables. You should be able to adjust the window size with the mouse until the plots look nice. Since you will also need to be able to embed these sort of pictures in documents take this figure and place it in a file using the **pdf()** or **png()** command, or through the word processing package you use on a regular basis.

Laboratory Session : Part 6

2017-05-23

Overview

In this session you will be able to use if-then rules in a **fis** and develop a small **fis**.

Instructions

Having had some exposure to membership functions, we can now start to consider how to put these together in rules in a **fis** in R. To do this we are going to develop a small set of rules. We have an example system that gives advice on whether a bank should give a loan based on two things: period of employment and salary . We have conducted a knowledge acquisition exercise with an expert from which we have gleaned the following information:

1. The salaries under consideration range from £0 to £100,000.
2. The period of employment is their number of year they have been in their current job, ranging from 0 to 40 years.
3. The linguistic salary ('salary') has three labels, *low, medium* and *high* where *medium* is best represented by a trapezoidal membership function and *low* and *high* are 'shoulders'.
4. The linguistic period of employment ('period') is covered by five labels (*very short, short, medium, long* and *very long*) and are represented by triangular membership functions.
5. The decision takes three linguistic terms *no*, *maybe* and *yes*. *No* and *yes* are triangular whilst *maybe* is trapezoidal. The decision range is 0 to 100.

There are two rules initially:

1. If *salary* is *high* and *period* is *very long* then *decision* is *yes*.
2. If *salary* is *low* and *period* is *very short* then *decision* is *no*.

To add these rules to a fis we use addrule, as in the following:

```
rulelist = rbind(c(3,5,3,1,1), c(1,1,1,1,1))
fis= addrule(fis, rulelist)
```

where *fis* is the fis name and *rulelist* is a matrix. See the documentation to understand more how **addrule** works. Once you have created the required variables, terms and associated rules, you can use **showfis** to print out the fis and example the rules in a textual format, or you can use **fis$ruleList**, where fis is the name of your fis object.

Your tasks are to:

1. create the variables and plot them on the same figure
2. add the rules to your system and show the rules;

A 'complete ruleset' would consist of every combination of the terms of salary with all the terms of period; in this case this would be a set of 15 rules. In a system such as this, it is not necessary to have a complete ruleset, but you need to ensure that every combination of inputs (salary 0 to 100000, and period 0 to 40) produces sensible

output. If you have spare time put in more rules to try to create a realistic system that fulfills this criteria.

Laboratory Session : Part 7

2017-05-23

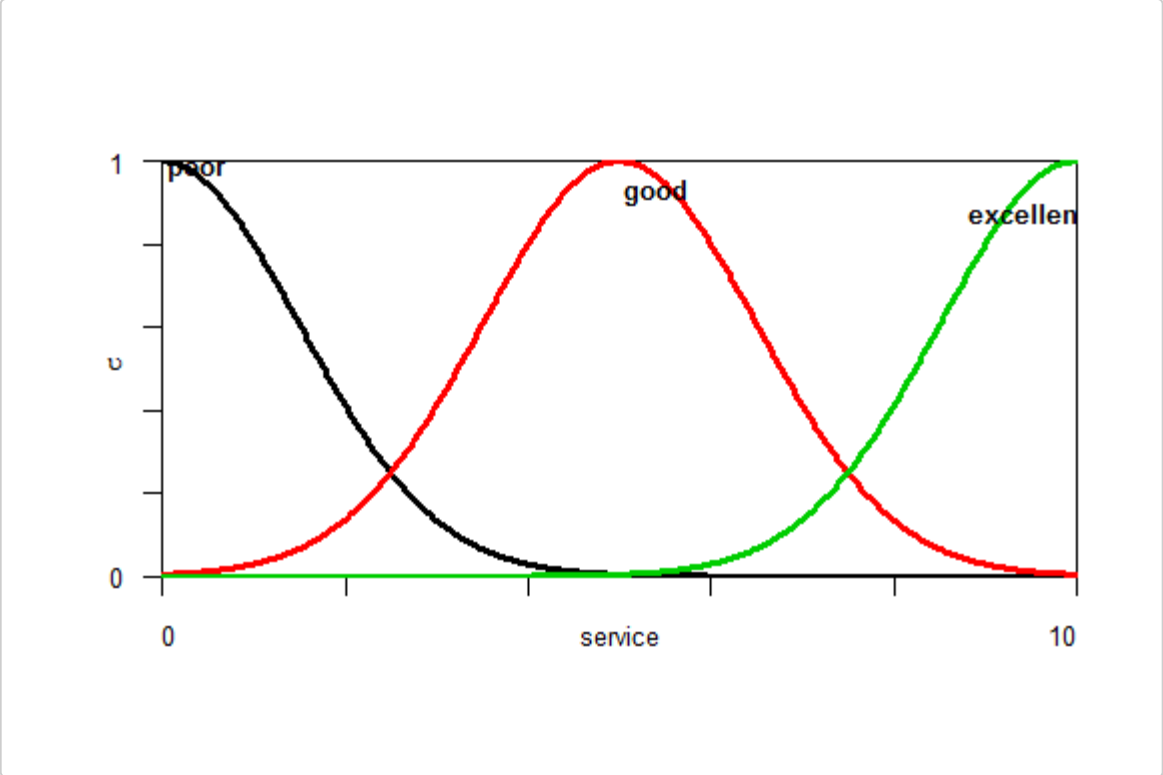
Instructions

You will now create the full restaurant tipping system in R, and run the fuzzy inference system to produce output.

Step 1

Enter the complete tipper fis as given in the MATLAB tutorial example (from the first lab!). Make sure you have entered it correctly by plotting the membership functions (all three on the same plot window), and using **showfis** to print out the fis and its rules.

```
plotmf(fis,"input",1)
```




```
showfis(fis)
```

```
## 1.  Name              tipper
## 2.  Type              mamdani
## 3.  Inputs/Outputs    [ 2 1 ]
## 4.  NumInputMFs       [ 3 2 ]
## 5.  NumOutputMFs      [ 3 ]
## 6.  NumRules          3
```

```
## 7.  AndMethod         min
## 8.  OrMethod          max
## 9.  ImpMethod         min
## 10. AggMethod         max
## 11. DefuzzMethod      centroid
## 12. InLabels          service
## 13. InLabels          food
## 14. OutLabels         tip
## 15. InRange           [ 0 10 ]
## 16. InRange           [ 0 10 ]
## 17. OutRange          [ 0 30 ]
## 18. InMFLabels        poor
## 19. InMFLabels        good
## 20. InMFLabels        excellent
## 21. InMFLabels        rancid
## 22. InMFLabels        delicious
## 23. OutMFLabels        cheap
## 24. OutMFLabels        average
## 25. OutMFLabels        generous
## 26. InMFTypes         gaussmf
## 27. InMFTypes         gaussmf
## 28. InMFTypes         gaussmf
## 29. InMFTypes         trapmf
## 30. InMFTypes         trapmf
## 31. OutMFTypes         trimf
## 32. OutMFTypes         trimf
## 33. OutMFTypes         trimf
## 34. InMFParams        [ 1.5 0 ]
## 35. InMFParams        [ 1.5 5 ]
## 36. InMFParams        [ 1.5 10 ]
## 37. InMFParams        [ 0 0 1 3 ]
## 38. InMFParams        [ 7 9 10 10 ]
## 39. OutMFParams       [ 0 5 10 ]
## 40. OutMFParams       [ 10 15 20 ]
## 41. OutMFParams       [ 20 25 30 ]
## 42. Rule Antecedent   [ 1 1 ]
## 43. Rule Antecedent   [ 2 0 ]
## 44. Rule Antecedent   [ 3 2 ]
## 45. Rule Consequent   1
## 46. Rule Consequent   2
## 47. Rule Consequent   3
## 48. Rule Weight       1
## 49. Rule Weight       1
## 50. Rule Weight       1
## 51. Rule Connection   2
## 52. Rule Connection   1
## 53. Rule Connection   2
```

Step 2

To actually run the fis on a set of inputs to produce an output we use the **evalfis** function. In simplest form, we can use it to produce a single output for one set of inputs (i.e. one value of service and food). For example:

```
evalfis(c(1,2),fis)
```

```
##          [,1]
## [1,] 5.558586
```

Step 3

Several inputs can be evaluated at once by putting each set of inputs into a separate row of an input matrix, using any standard R method, for example:

```
inputs= rbind(c(3,7),c(2,7))
evalfis(inputs,fis)
```

```
##           [,1]
## [1,] 12.218379
## [2,]  7.788532
```

Step 4

The return value of **evalfis** is a single matrix of output values, so this may be put into a variable:

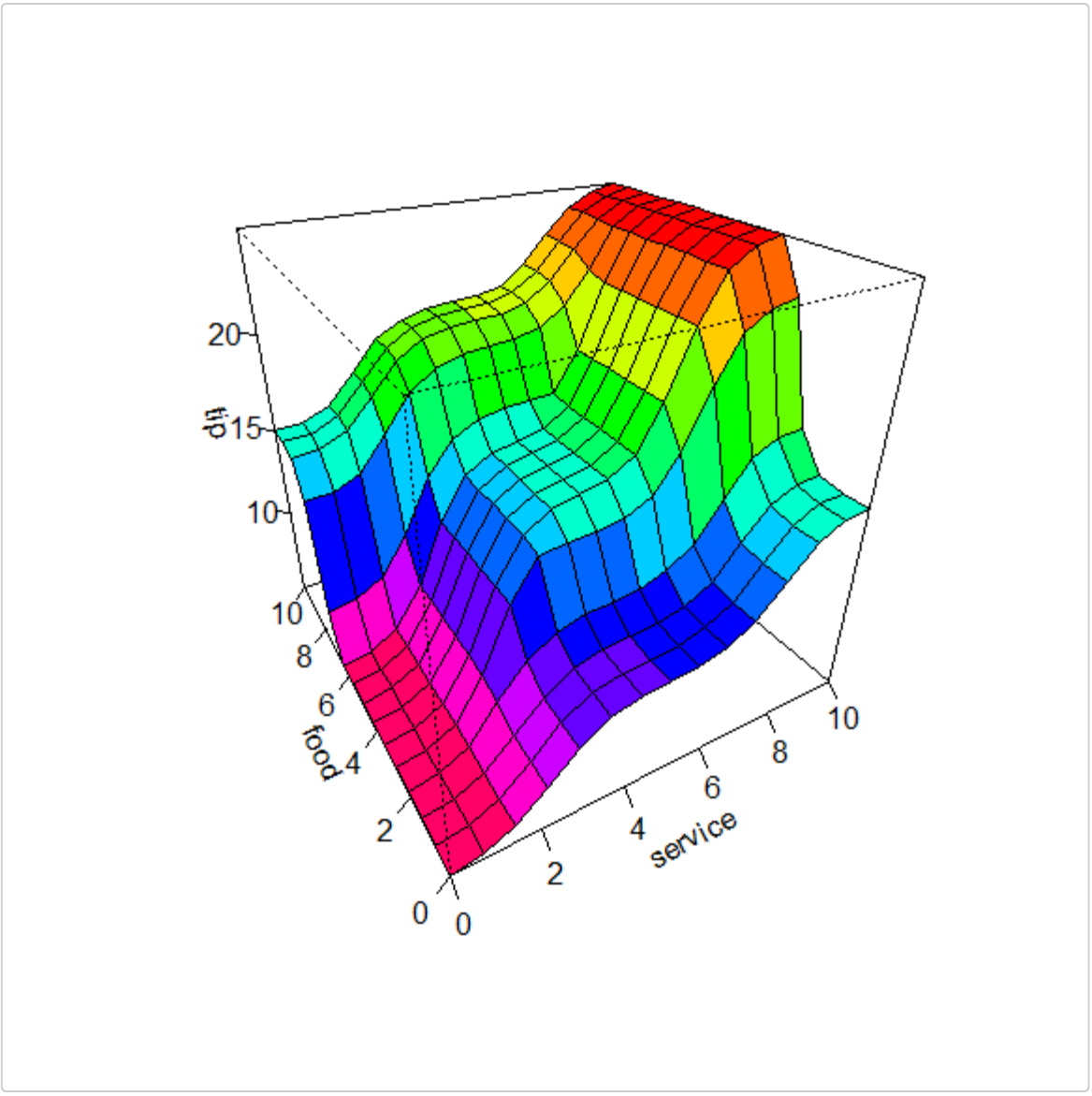
```
outputs= evalfis(inputs,fis)
# display the value of variable outputs
outputs
```

```
##           [,1]
## [1,] 12.218379
## [2,]  7.788532
```

Step 5

R provides a useful function to visualise the output surface plot for a two-input-one-output fis:

```
gensurf(fis)
```



Note! If your 3D plot is very small, run the following command to reset the way graphs are drawn:

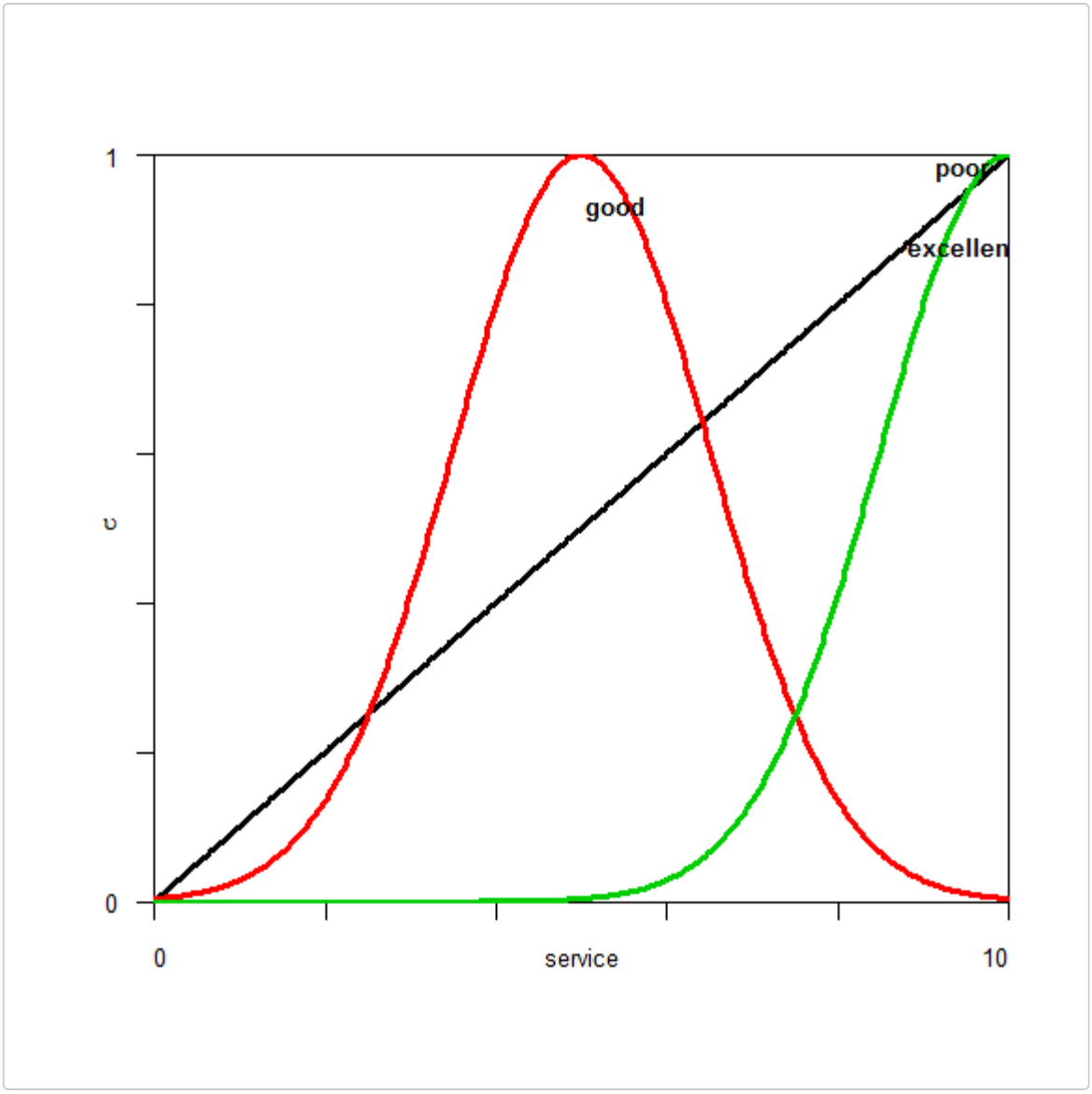
```
par(mfrow=c(1,1))
```

Try different values of membership functions, and different rules sets to alter the output surface of the tipper example.

Step 6

Finally, you may notice that the built-in fis functions only allow you to **addmf**, not to (for example) **delmf** or **setmf**. This means that you have to keep creating a new fis each time you want to change even one parameter of one membership function. To avoid this, you can use standard R commands to alter the fis structures directly, such as:

```
fis$input[[1]]$mf[[1]]$type= "trimf"
fis$input[[1]]$mf[[1]]$params= c(0,10,10,1)
plotmf(fis,"input",1)
```


1
0
c
good
poor
excellen
0
service
10

Graphic User Interface (GUI) with FuzzyR

2017-05-23

Overview

A GUI is developed similar to that of Matlab? Fuzzy Logic Toolbox which will be more user friendly and more interactive. This GUI enable the user to view every elements of fuzzy logic systems such as membership function (MFs) graph, rules and to evaluate the fuzzy inferences.

Load the package

First, load the package:

```
library(FuzzyR)
library(shiny)
```

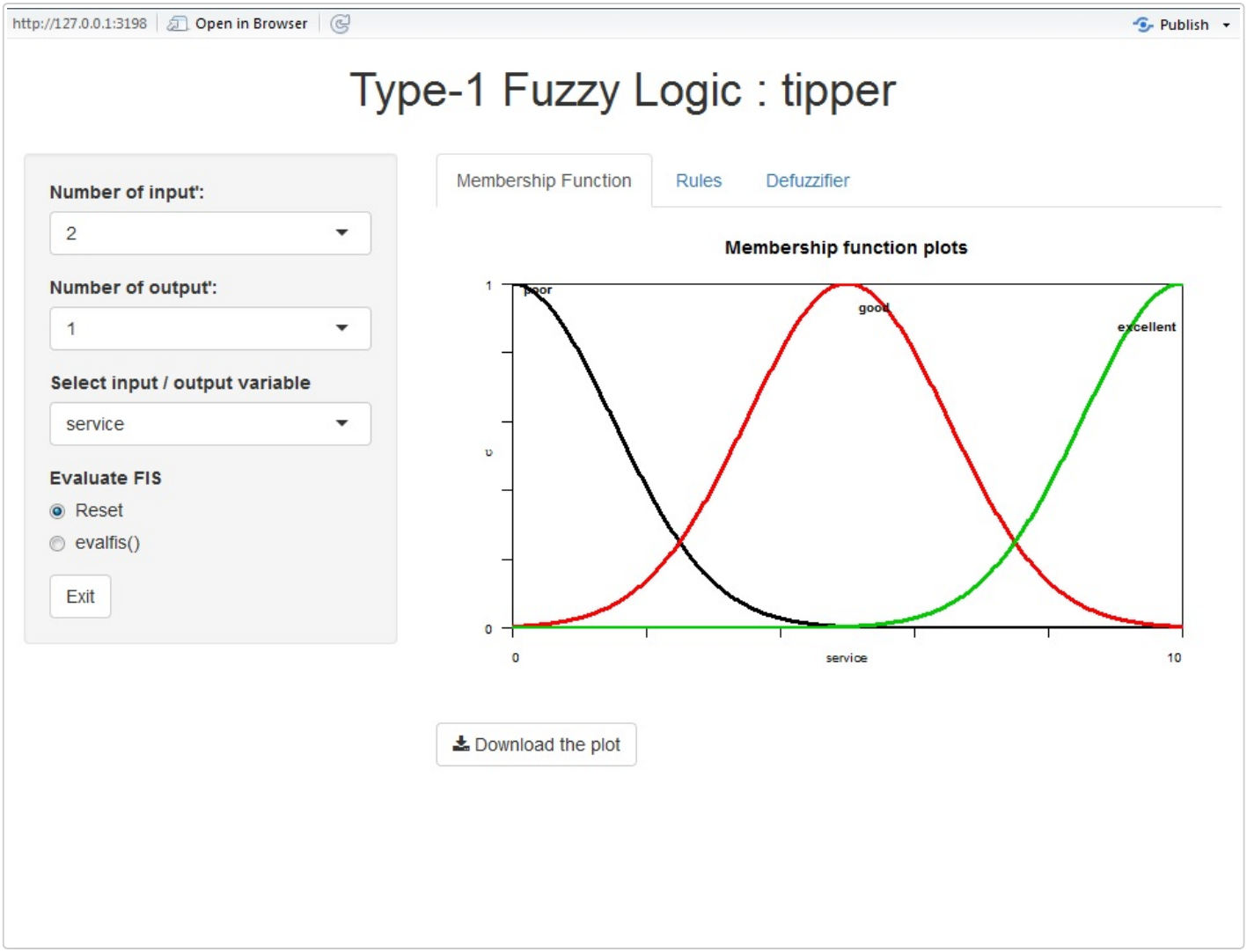
Write and display GUI of a Waiter-Tipping example

Then, use a *showGUI()* function to display a Waiter-Tipping example:

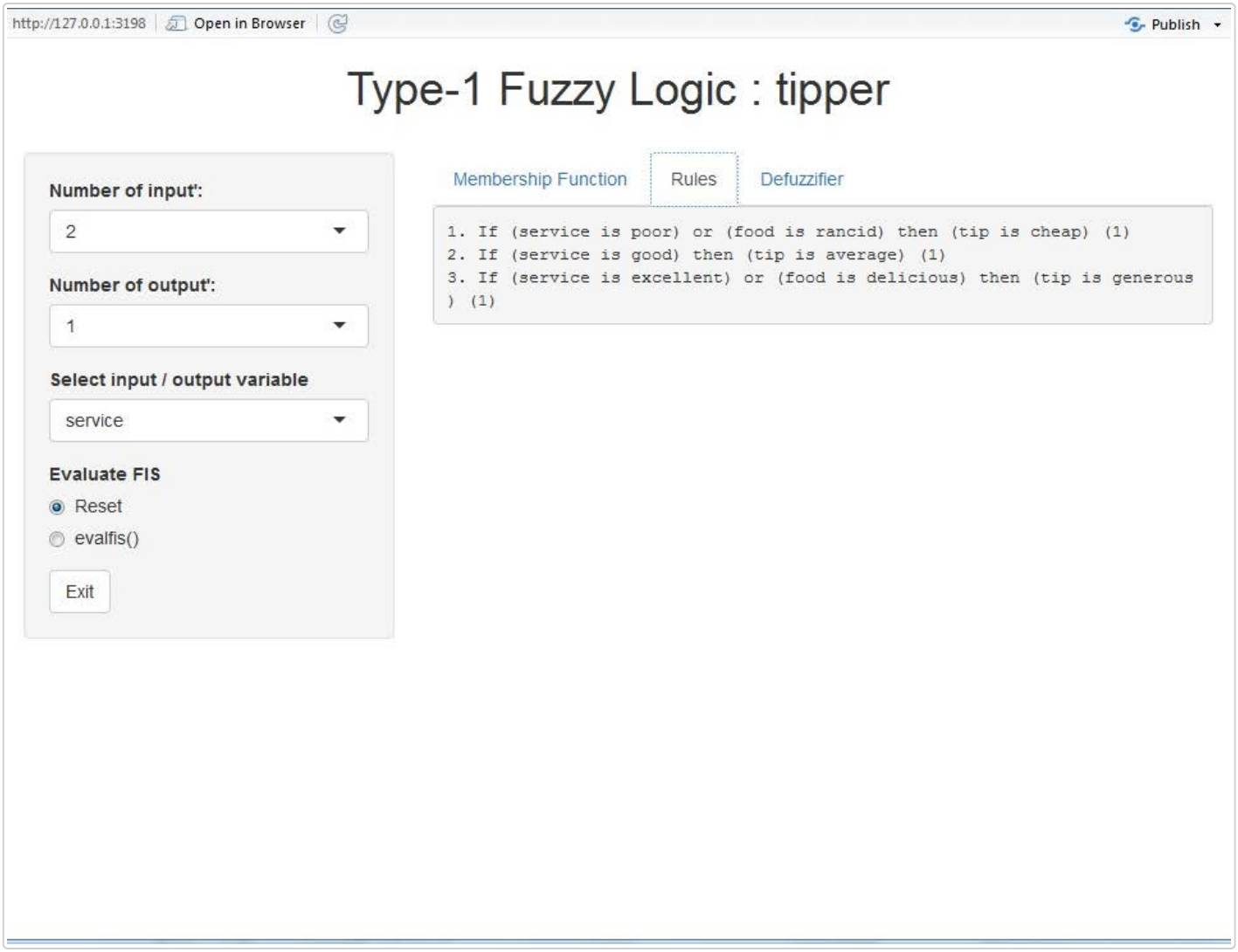
```
library(shiny)
# Creates a fis object.
fis <- newfis('tipper')

# Adds an input or output variable to a fis object.
fis <- addvar(fis, 'input', 'service', c(0, 10))
fis <- addvar(fis, 'input', 'food', c(0, 10))
fis <- addvar(fis, 'output', 'tip', c(0, 30))

# Adds a membership function input 1 : service
fis <- addmf(fis, 'input', 1, 'poor', 'gaussmf', c(1.5, 0))
fis <- addmf(fis, 'input', 1, 'good', 'gaussmf', c(1.5, 5))
fis <- addmf(fis, 'input', 1, 'excellent', 'gaussmf', c(1.5, 10))

# Adds a membership function input 2 : food
fis <- addmf(fis, 'input', 2, 'rancid', 'trapmf', c(0, 0, 1, 3))
fis <- addmf(fis, 'input', 2, 'delicious', 'trapmf', c(7, 9, 10, 10))

# Adds a membership function output 1 : tip
fis <- addmf(fis, 'output', 1, 'cheap', 'trimf', c(0, 5, 10))
fis <- addmf(fis, 'output', 1, 'average', 'trimf', c(10, 15, 20))
fis <- addmf(fis, 'output', 1, 'generous', 'trimf', c(20, 25, 30))


# Adds a rule to a fis object.
```

```
rl  <- rbind(c(1,1,1,1,2), c(2,0,2,1,1), c(3,2,3,1,2))
fis <- addrule(fis, rl)

# Show a GUI to display MFs plots for input and output, rules and evaluate the fis.
showGUI(fis)
```

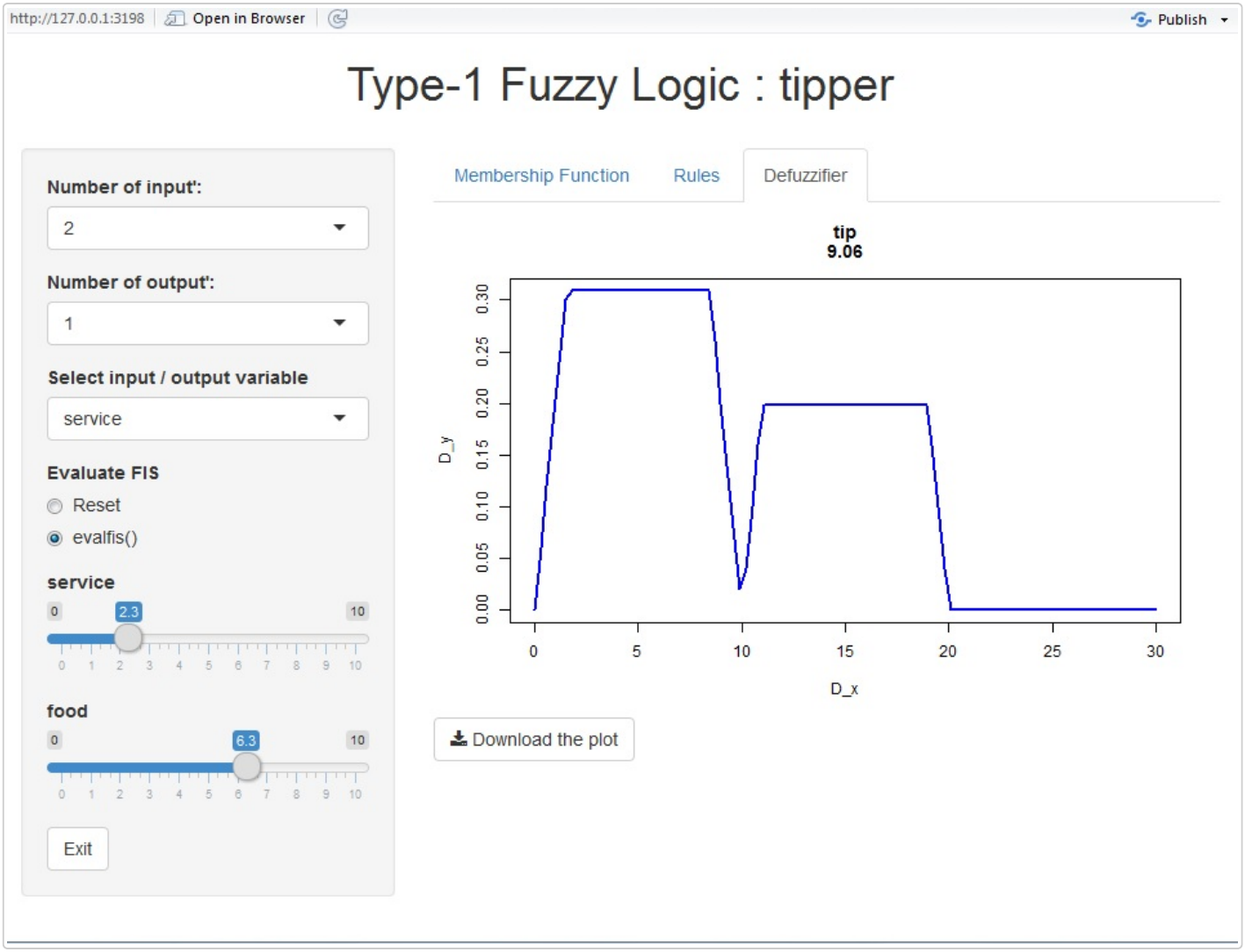
GUI of a Waiter-Tipping:



- View all membership functions (MFs) the from the specified variable which must be part of a Waiter-Tipping system. You can also download a plot for each Mfs.



- Show the total of rules inside Waiter-Tipping system.



- Evaluate a Fuzzy Inference System (fis) of Waiter-Tipping system.